

COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055



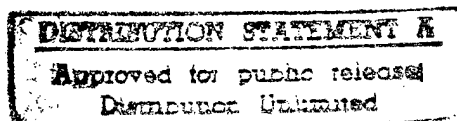
An Ada-Prolog System

Neel Madhav

Technical Report No. CSL-TR-90-437

Program Analysis and Verification Group Report No. 49

August, 1990



This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

DTIC QUALITY INSPECTED 3

19960916 152

CLEARED
FOR OPEN PUBLICATION

SEP 09 1996

96-5-3849
DIRECTORATE FOR FREEDOM OF INFORMATION
AND SECURITY REVIEW (OASD-PA)
DEPARTMENT OF DEFENSE

96-5-3849



OFFICE OF THE UNDER SECRETARY OF DEFENSE (ACQUISITION & TECHNOLOGY)
DEFENSE TECHNICAL INFORMATION CENTER
8725 JOHN J KINGMAN RD STE 0944
FT BELVOIR VA 22060-6218



IN REPLY
REFER TO

DTIC-OMI

1 AUG 96

SUBJECT: Distribution Statements on Technical Documents

TO:

U.S. DEFENSE ADVANCED RESEARCH PROJECT
AGENCY/INFORMATION SYSTEMS OFFICE
3701 NORTH FAIRFAX DRIVE
ARLINGTON, VA 22203-1714

1. Reference: DoD Directive 5230.24, Distribution Statements on Technical Documents, 18 Mar 87.
2. The Defense Technical Information Center received the enclosed report (referenced below) which is not marked in accordance with the above reference.

"AN ADA-PROLOG SYSTEM" REPORT #CSL-TR-90-437 CONTRACT #N00039-84-C-0211

3. We request the appropriate distribution statement be assigned and the report returned to DTIC within 5-working days.
4. Approved distribution statements are listed on the reverse of this letter. If you have any questions regarding these statements, call DTIC's Input Support Branch, (703) 767-9092, 9088 or 9086 (DSN use prefix 427).

FOR THE ADMINISTRATOR:

1 Encl

Crystal Riley
CRYSTAL RILEY
Chief, Input Support Branch

FL-171
Dec 95

DoD Directive 5230.24, "Distribution Statements on Technical Documents," 18 Mar 87, contains seven distribution statements, as described briefly below. Technical Documents that are sent to DTIC must be assigned one of the following distribution statements:

☒ **DISTRIBUTION STATEMENT A:**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

☐ **DISTRIBUTION STATEMENT B:**

DISTRIBUTION AUTHORIZED TO U. S. GOVERNMENT AGENCIES ONLY; (FILL IN REASON); (DATE STATEMENT APPLIED). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (INSERT CONTROLLING DoD OFFICE).

☐ **DISTRIBUTION STATEMENT C:**

DISTRIBUTION AUTHORIZED TO U. S. GOVERNMENT AGENCIES AND THEIR CONTRACTORS; (FILL IN REASON); (DATE STATEMENT APPLIED). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (INSERT CONTROLLING DoD OFFICE).

☐ **DISTRIBUTION STATEMENT D:**

DISTRIBUTION AUTHORIZED TO DoD AND DoD CONTRACTORS ONLY; (FILL IN REASON); (DATE STATEMENT APPLIED). OTHER REQUESTS SHALL BE REFERRED TO (INSERT CONTROLLING DoD OFFICE).

☐ **DISTRIBUTION STATEMENT E:**

DISTRIBUTION AUTHORIZED TO DoD COMPONENTS ONLY; (FILL IN REASON); (DATE STATEMENT APPLIED). OTHER REQUESTS SHALL BE REFERRED TO (INSERT CONTROLLING DoD OFFICE).

☐ **DISTRIBUTION STATEMENT F:**

FURTHER DISSEMINATION ONLY AS DIRECTED BY (INSERT CONTROLLING DoD OFFICE AND DATE), OR HIGHER DoD AUTHORITY.

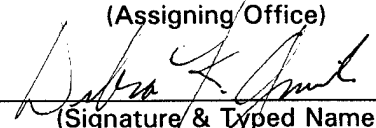
☐ **DISTRIBUTION STATEMENT X:**

DISTRIBUTION AUTHORIZED TO U. S. GOVERNMENT AGENCIES AND PRIVATE INDIVIDUALS OR ENTERPRISES ELIGIBLE TO OBTAIN EXPORT-CONTROLLED TECHNICAL DATA IN ACCORDANCE WITH DoD DIRECTIVE 5230.25 (DATE STATEMENT APPLIED). CONTROLLING DoD OFFICE IS (INSERT).

(Reason)

OASB/TIO

(Assigning Office)


(Signature & Typed Name)

Debra K. Amick

Defense Advanced Research Projects Agency (DARPA)
(Controlling DoD Office Name)

3701 North Fairfax Drive, Arlington, VA 22203
(Controlling DoD Office Address (City/State/Zip))

September 9, 1996

(Date Statement Assigned)

An Ada—Prolog System

Neel Madhav*

Program Analysis and Verification Group
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Computer Systems Laboratory Technical Report CSL-TR-90-437

Program Analysis and Verification Group Report No. 49

August, 1990

Abstract

This paper presents a *software development tool* — the Ada-Prolog system which combines the strengths of both descriptive and procedural programming styles. Concrete reasons and examples are provided to demonstrate that such a tool would be useful.

This tool provides various operations available in Prolog for clause building, database building and querying to Ada programs. *In addition* to allowing dynamic access to both Ada and Prolog, the Ada-Prolog system adds to the functionality provided by Prolog by *partitioning* the Prolog database into *lists* of clauses. These lists can be created, updated and destroyed dynamically. *Concurrent* access to the list of clauses is also possible. Queries can be directed to groups of these lists.

The system is meant for use in expert systems, compilers, database applications, rapid prototyping systems, advanced environments, and other software tools which use deduction.

Keywords — *Ada, knowledge bases, logic programming, procedural languages, Prolog, software engineering, specification, specification analysis.*

*Computer Science Department, Stanford University, Stanford, CA 94305. E-mail: Madhav@Cs.Stanford.Edu

Computer Systems Laboratory
Stanford University
Copyright © 1990

1 Introduction

There is an increasing proliferation of environments and applications which use both *procedural* and *descriptive* programming techniques [2,17]. These systems need capabilities to store and reason with knowledge as well as capabilities like abstract interface specification, efficient description of algorithms and concurrent execution available in general-purpose procedural languages.

Prolog is a descriptive language used mainly for symbolic computation [7]. Prolog's declarative style and use of logical inference make it ideal for implementing expert system inference engines and knowledge bases [3,21]. Ada is a general-purpose procedural language used to implement programs from a wide variety of computational paradigms [1]. Applications of Ada range from programming environments to real-time systems.

This paper presents a *software development tool* — the Ada-Prolog system that draws upon the strengths of both procedural and descriptive programming styles to provide access to these different programming paradigms in the same system. Figure 1 shows various properties of Ada and Prolog and the fact that the Ada-Prolog system provides a *cross product* of these properties.

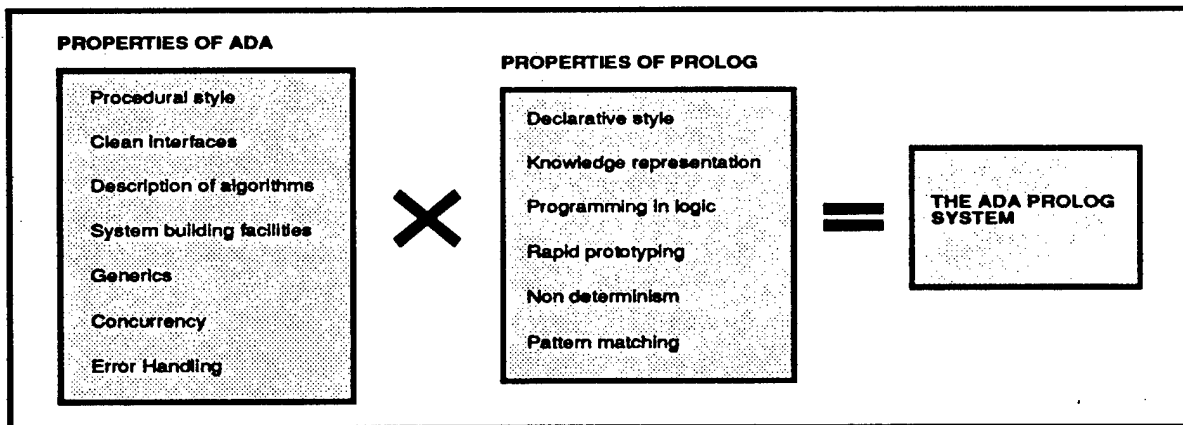


Figure 1: The Ada-Prolog System Provides a Cross-Product of the Properties of Ada and Prolog

The four main contributions of this system are—1) combination of two different, complementary programming styles, 2) adding software engineering programming techniques and concurrency available in Ada to Prolog, 3) adding a capability to partition the database to Prolog, and 4) adding symbolic reasoning to Ada.

There are other systems which seek to combine Prolog and other languages to provide two different programming paradigms in one system. Combinations of Lisp and Prolog which combine functional and logic programming are QLOG [10], LOGLISP [18], QUTE [19], PiL [22] and various other systems. These systems seek to combine the syntax of Prolog and Lisp and allow Prolog and Lisp expressions to be embedded in each other. PiL provides *chunking* of the Prolog database, where a user can distribute the database among sub-databases. A system which seeks to combine Modula-2 and Prolog is [15]. Numerous systems combine Prolog and

various database management systems [11]. One example of this is [14].

Instead of trying to force the two different styles of Ada and Prolog to live together and combining Ada and Prolog *as languages* (and produce one unified language), the system provides access to both Ada and Prolog in a unified framework. This allows a user of the system to program in Ada and Prolog in close co-operation without diluting the strengths of either language. See [16] for a full discussion of why different languages in mixed language systems should not be unified into one language. Such systems, it is argued, end up compromising the strengths of their constituent languages without tangible benefits. Programmers familiar with either language do not find a comfortable environment to work with.

In *addition* to the facilities provided by the systems mentioned above, the Ada-Prolog system provides the rich facilities of Ada, and also a capability to partition the Prolog database. This facility for partitioning the Prolog database combined with other features of Ada allows the formation of constructs like modules, types, and generic units in Prolog [9]. The lack of modularity in Prolog is one of the difficult problems with Prolog applications [4]. The syntactical approach to modularization adds modules as a syntactic construct to Prolog [8]. Another approach is to introduce modularization in the environment [5]. The Ada-Prolog system takes the second approach and provides modularization through partitioning of the Prolog database.

The concurrency provided by the Ada-Prolog system is at the query level. That is, two queries at the *top level* can proceed concurrently with each other but there is no concurrency between sub-queries *within* a query. The Ada-Prolog system does not provide the concurrency available in PARLOG [6] or Concurrent Prolog [20].

The Ada-Prolog system is an Ada program library package. This package provides logic programming operations to Ada programs. In addition to the facilities of both Prolog and Ada, the Ada-Prolog system has a facility for dynamically constructing separate lists of clauses instead of a single database. Section 2 discusses the Ada-Prolog system.

This package has been completely specified using Anna [12], a language for specifying Ada programs. This formal specification of the system allows the user to understand the functionality of the Ada-Prolog system at an abstract level without the need to understand the implementation.

Section 3 presents a simple example which demonstrates the capabilities of the Ada-Prolog system. Section 4 describes applications of the interface. An environment tool developed using the Ada-Prolog system is also outlined. Section 5 concludes the paper.

2 The Ada — Prolog System

The interface to the Ada-Prolog system is an *abstractly specified package which provides logic programming operations to Ada programs*. At the heart of the implementation of this package is a modified Prolog interpreter written in Ada.

There are two basic activities taking place in a Prolog system:

- *defining facts* and relationships between facts, called *rules* and
- *querying* whether certain facts follow from other facts and relationships

These activities can be looked upon as the process of building a *knowledge base* and then using an *inference engine* to draw *inferences* from it. The Ada-Prolog system provides two additional facilities:

- dynamically creating, changing and deleting *databases* of facts and rules.
- building systems with abstract interfaces which can execute queries *concurrently*.

This does not change the logical aspects of Prolog but allows the existence of multiple databases. As we shall demonstrate later, this facility is of great practical use.

The Ada-Prolog system provides abstract operations for all of the above activities. There are three broad parts of the Ada-Prolog system interface—a *clause building* sub-package, a *database* sub-package and a *query* sub-package.

2.1 Clauses

Clauses are used to represent Prolog *structures*. That is, they form both the assertion language and the query language of the system.

Clauses are represented as abstract trees. Each internal node of the tree is a constant which represents the *functor* of the clause. Each of the leaves of the tree is a constant, a variable or a number. Numbers can be looked upon as constants which cannot have children. Clauses can be *facts* and *rules*. Rules differ from facts in that they have a head and a tail. Figure 2 shows the structure of a clause. The structure is shown in a BNF like notation. Vertical bars denote alternation. Stars denote zero or more occurrences of the preceding construct. Clauses are a

```

<Clause>    ::= <Fact> | <Rule> ;
<Fact>      ::= <Constant> | <Predicate> ;
<Rule>      ::= <Head> <Tail> ;
<Predicate> ::= <Constant> ( <Term> ) * ;
<Head>      ::= <Fact> ;
<Tail>      ::= ( <Fact> ) * ;
<Term>      ::= <Constant> | <Variable> |
               <Number> | <Fact> ;

```

Figure 2: The Structure of Clauses

private type in the Ada-Prolog system. The internal representation of these types is not visible to the user. Constants, variables and numbers are provided as primitives in the Ada-Logic system. Various clause building operations are provided.

It is sometimes cumbersome to build clauses by adding nodes to trees. Therefore the Prolog parser has been isolated and is provided as a routine at the interface level. This routine, given a string, parses it and returns a clause. All these routines follow Prolog syntax conventions. Another routine, given a clause dumps the clause into a string.

Figure 3 shows a part of Ada-Prolog that deals with building and selecting parts of clauses.


```

type CLAUSE is private;
function BUILD_FACT(ATTR : INTEGER) return CLAUSE;
function BUILD_FACT(ATTR : IDENTIFIER)
    return CLAUSE;
-- Fact building operations.
...
function IS_RULE(C : CLAUSE) return BOOLEAN;
function BUILD_RULE(HEAD_OF_CLAUSE : CLAUSE;
    TAIL : LIST_OF_SONS) return CLAUSE;
function GET_HEAD(C : CLAUSE) return CLAUSE;
function GET_TAIL(C : CLAUSE) return LIST_OF_SONS;
-- Rule disassembling and rule building operations.
...
-- Other operations.
function READ_CLAUSE(TEXT : STRING) return CLAUSE;
function DUMP_CLAUSE(TREE : CLAUSE) return STRING;

```

Figure 3: Some Operations on Clauses

2.2 Databases

The Ada-Prolog system allows the Prolog database to be partitioned into *lists of clauses*. This is useful for three reasons. Firstly, this reduces the search space for queries since it allows the user to keep different kinds of information separate. For example, knowledge about airline schedules can be kept separate from knowledge about airline employee work hours. Secondly, different users can share a list, possibly concurrently, keeping their own data safely separate from other users. Different users can thus communicate through the Ada-Prolog system. Thirdly this partitioning combined with Ada scoping allows the introduction of *scoping* in the Prolog database. A user can put assertions into a list and test their consequences *before* committing to that list.

A list of clauses is indexed by integers. These lists can store an arbitrary number of clauses and can be created, destroyed and modified *dynamically*. Lists of clauses are implemented by a time and space efficient data structure. Hashing techniques are used to achieve time efficiency and shared data structures are used to achieve space efficiency. A reference on Prolog implementations is [23]. There is, in addition, a global database which has Prolog system information. This global database is also expected to contain user defined, system wide information.

One would need, in general, to direct queries to more than one list. For example, a compiler might want to keep information about integers, numeric types, and general types in separate lists but might want to direct queries to all of the lists together. Therefore, *lists of lists* are defined. Lists of clauses can be grouped into lists of lists. A list of clauses can belong to more than one list of lists. Queries can then be directed to these lists of lists.

Operations are available to read a list of clauses, written out in Prolog syntax, from a file. An inverse operation that writes lists of clauses onto files is also provided.

Figure 4 shows the section of the Ada-Prolog package which deals with lists of clauses and lists of lists.

```

type LIST_OF_CLAUSES is private;
function CREATE return LIST_OF_CLAUSES;
function APPEND_CLAUSE(C : CLAUSE;
                      L : LIST_OF_CLAUSES)
    return LIST_OF_CLAUSES;
-- Operations to build and disassemble lists of clauses.
...
type LIST_OF_LISTS is private;
function CREATE return LIST_OF_LISTS;
function APPEND_LISTS(L : LIST_OF_CLAUSES;
                     LL : LIST_OF_LISTS)
    return LIST_OF_LISTS;
-- Operations to build and disassemble lists of lists.
...
function READ_FILE(F : FILE_TYPE)
    return LIST_OF_CLAUSES;
procedure WRITE_FILE(L : LIST_OF_CLAUSE;
                     F : FILE_TYPE);
-- File Operations.

```

Figure 4: Database Operations

2.3 Queries

Queries, posed in the form of a clause with unbound variables, can be directed to lists of clauses or lists of lists. An answer to a query is, either *true* with a list of substitutions of free variables which make the query true, or the answer is *false*.

There are two modes of query available in the Ada-Prolog system. A user can get all possible answers at one time or get the answers one at a time. Figure 5 shows the section of the Ada-Prolog package which deals with queries.

The type *answer* is a private type which is a list of variable-clause pairs. These pairs represent the bindings of free variables in a query. The data-structure *list of answers* is used to return all answers to a query.

2.3.1 Concurrent Execution of Queries

Since Ada has facilities for concurrent execution of programs, different users can direct queries concurrently to the Ada-Prolog system. Placing a query inside a separate thread of control through a *task* allows the query to proceed concurrently with other queries. If there was no mechanism for sharing lists which the queries act upon, the concurrency provided would not be powerful enough to model most concurrent systems of today. Therefore, different queries may share a common list of clauses. Figure 6 shows concurrent execution of queries.

To exclude the possibility of a concurrent update of lists, a very simple *locking* facility is provided. Shared lists may be queried concurrently, a query may not proceed concurrently with an update and two updates may not proceed concurrently.

```

type ANSWER is private;
type LIST_OF_ANSWERS is private;
...
-- Operations to disassemble answers.
...
procedure QUERY(QUESTION : CLAUSE;
                BASED_ON : LIST_OF_LISTS;
                SUCCESS : out BOOLEAN;
                BINDINGS : out ANSWER);
-- Return one possible set of bindings.
procedure NEXT_ANSWER(SUCCESS : out BOOLEAN;
                     BINDINGS : out ANSWER);
-- Get the next answer for the previously asked query.
procedure QUERY(QUESTION : CLAUSE;
                BASED_ON : LIST_OF_LISTS;
                SUCCESS : out BOOLEAN;
                SOLUTIONS : out LIST_OF_ANSWERS);
-- Return all possible answers to a query.

```

Figure 5: Query Operations

3 A Simple Example: A Family Tree Knowledge Base

This example illustrates combination of procedural and descriptive styles, using various facilities of Ada and Prolog, and partitioning of the Prolog database.

This example outlines an implementation of a shared knowledge base of family trees. Family trees will be stored as Prolog facts of the form *parent(X,Y)*. This stands for the fact that *X* is a parent of *Y*. A user can *create* a new tree, *modify* a tree or ask *queries* about a family tree. All these actions will be performed concurrently with other users. Queries may be about relationships satisfied by a tree. For example, a typical query could be *ancestor(X,Y)*, whether *X* is an ancestor of *Y*. Rules for inferring relationships are common to all families.

An *implementor* has to provide all these facilities in a user-friendly manner as well as implement the knowledge base such that rules about new relationships can be added to the knowledge base dynamically.

We now describe an implementation of the knowledge base.

Each family will have its tree in a separate list. Since new lists can be created and modified dynamically, creating new family trees and modifying family trees is just a matter of a subprogram call to the Ada-Prolog system. Notice that if names in different families are the same, this does not create a problem. Also, updates to a family tree can be done independent of other family trees. Since lists can be modified and queried concurrently, this is simple to implement in the Ada-Prolog system. Figure 7 shows part of the code used to implement family trees.

Storing and retrieving family trees from files is quite simply implemented by calls to routines in the Ada-Prolog system interface. Notice that if one needs to merge two family trees, as one might when two people get married, this too is just a matter of a subroutine call. Figure 8 shows part of the code to change, merge and update family trees.

Common knowledge, like rules for inferring whether *X* is an ancestor of *Y*, are stored in a

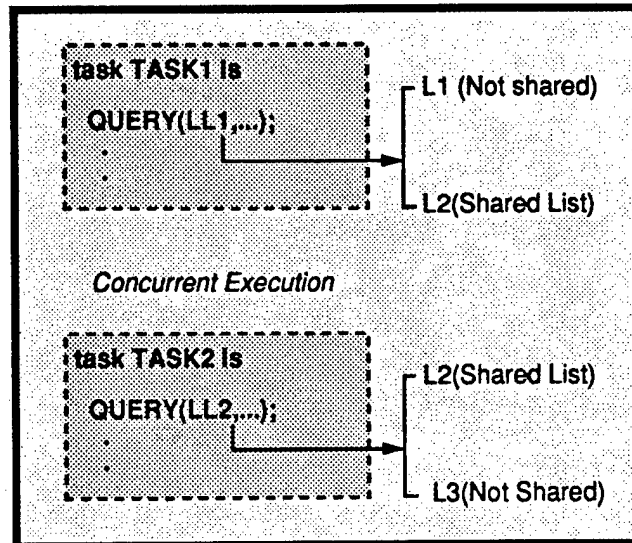


Figure 6: Concurrently Executing Queries Can Share a List

```
...
SMITH_FAM_TREE : LIST_OF_CLAUSES;
SMITH_FAM_TREE := CREATE;
-- Create a tree.
APPEND_CLAUSE(READ_CLAUSE("parent(john, liz)"),
              SMITH_FAM_TREE);
-- Add various connections to the family tree. Facts can be built in three ways:
-- by building the fact node by node, by using the parsing routines of the system
-- to parse a string representing the fact and by reading the fact from a file.
```

Figure 7: Creating Family Trees

shared list. Rules can be added or deleted from this list dynamically. The Prolog inference engine acts as the inference engine for this system. All queries about the knowledge base are answered through this inference engine. Since queries can be directed to arbitrary sets of lists, queries can be directed to the union of a family tree and the common knowledge base. Figure 9 illustrates the insertion of rules into this common database.

Queries can now be directed to the system about various family trees. These queries can be concurrently directed to the same lists but lists cannot be updated concurrently with queries. Figure 10 illustrates part of the code which may be used to query the system.

The user interface of the system can be implemented in Ada. There are numerous windowing and graphics facilities available in Ada for this purpose. All the input and output is done through Ada routines. All the commands and queries go through Ada. Therefore the control of the whole knowledge base is in Ada. Calls are made to Prolog whenever appropriate. If one needs to code some algorithmic knowledge as part of the knowledge base, for example sorting all members of a family tree, this can be implemented simply and elegantly in Ada.

```

APPEND_CLAUSE(READ_CLAUSE("parent(john, mary)"),
              SMITH_FAM_TREE);
-- Adding a clause to a tree.
REMOVE_CLAUSE(READ_CLAUSE("parent(X,liz)"),
              SMITH_FAM_TREE);
-- Removing a clause from the family tree.
NEW_FAMILY_TREE := APPEND_LISTS(SMITH_FAM_TREE,
                                JOHNSON_FAMILY_TREE);
-- Merging two family trees.

```

Figure 8: Updating and Manipulating Family Trees

```

HEAD := READ_CLAUSE("ancestor(X,Y)");
TAIL := READ_CLAUSE("parent(X,Y)");
TAIL_LIST := CREATE;
APPEND_CLAUSE(TAIL_LIST, TAIL);
ANCESTOR_RULE := BUILD_RULE(HEAD, TAIL_LIST);
-- Creating a rule. Rules can also be created in all three ways for creating
-- facts.
APPEND_CLAUSE(GLOBAL_LIST, ANCESTOR_RULE);
-- Adding a rule to the global list.

```

Figure 9: Creating the Global Database

```

...
COMBINED_SMITH_LIST :=
APPEND_LISTS(GLOBAL_LIST, SMITH_FAM_TREE);
COMBINED_JOHNSON_LIST :=
APPEND_LISTS(GLOBAL_LIST, JOHNSON_FAMILY_TREE);
-- Combine databases.
Q := READ_CLAUSE("ancestor(X, john)");
-- Formulate a query.
QUERY(Q, COMBINED_SMITH_LIST, SUCC, ANSWERS);
-- Pose the query. The variable SUCC will be true if there is an ancestor of
-- john. All the successors of john, if any, will be in the list variable AN-
-- SWERS. Routines are provided for disassembling answers.
...
task body TASK1 is
...
QUERY(Q, COMBINED_JOHNSON_LIST, SUCC, ANSWERS2);
...
end TASK1;
-- The task executes in a separate thread of control. The two queries, thus
-- execute concurrently. The list GLOBAL_LIST is shared by the two queries.
-- If there was a need for communication between the two queries, they could
-- communicate through adding assertions to a common list of clauses.

```

Figure 10: Querying the Family Tree System

4 Ada-Prolog Applications

Applications for the Ada-Prolog system can be divided into two categories—applications traditionally implemented in Ada but would benefit from the strengths of Prolog, and applications traditionally implemented using Prolog but would benefit from the strengths of Ada. The first kind of applications are advanced environments, compilers, and a variety of defense industry embedded applications. The second kind of applications are expert systems, certain database applications, rapid prototyping systems and theorem proving systems.

There are three levels at which Ada and Prolog complement each other. Firstly they have complementary styles of programming—procedural for Ada and descriptive for Prolog. Secondly, each language can use software written in the other language. Thirdly, each language can remedy the other's defects[16]. For example, Prolog has backtracking which Ada does not have and Ada has destructive assignment which Prolog does not have.

Ada provides facilities available in procedural languages like Pascal as well as a rich type structure, packages, concurrency and exception handling. Ada supports software engineering concerns by providing facilities like separate compilation, program libraries, clean interfaces and generic units. Ada programs have access to a vast amount of pre-existing software and tools. For example, parsers, editors, debuggers, databases, window systems, graphics tools and even operating systems are available in Ada. The advantage of Ada is that it is portable. Thus any Ada program can use the tools mentioned above. Applications traditionally implemented using Prolog can benefit from all these facilities that Ada provides.

The unique properties of Prolog facilitate rapid prototyping, knowledge representation, programming in logic, non-determinism, backtracking and pattern matching. Traditionally tools like compilers and databases were implemented in procedural languages like C. Artificial intelligence techniques are increasingly being used to incorporate expert system like functionality in these systems [3,2]. These systems and other advanced environment tools also need a capability for *deduction*. Prolog is eminently suited for providing these facilities. The Ada-Prolog system can provide expert system building facilities as well as deduction to programs traditionally implemented in procedural languages. An illustration of this is provided in figure 1.

As demonstrated in section 3 the partitioning capability is also very useful for both kinds of applications.

4.1 Implementation of a Specification Analyzer

The specification analyzer [13] uses the Ada-Prolog system to *symbolically execute program specifications* written in Anna, to verify their properties *before* an implementation is provided.

The specification analyzer can be looked upon as a number of co-operating systems. These systems specialize in symbolic execution of programs, typing rules of Ada, and theorem proving. Each of these systems assists in symbolic execution of program specifications. The typing rules are needed to ensure a correct specification, the symbolic execution of programs allows parts of the specification to be executed, and the theorem prover allows the system to prove certain properties of specifications.

These systems share a Prolog database through the Ada-Prolog system. This database is hierarchically organized with more basic knowledge stored at a lower level and complex knowl-

edge stored at a higher level. Each of these systems has its own inference engine implemented in Prolog. The common database is also implemented in Prolog. The user interface of the specification analyzer is implemented in Ada. Ada is also used to implement portions of the specification analyzer that require an algorithmic style of programming, for example, the Anna parser and the X windows interface to the specification analyzer.

5 Conclusion

The Ada-Prolog system is a software tool which has wide application. The system combines the strengths of both Ada and Prolog to provide a flexible environment which allows both descriptive and procedural styles of programming.

For applications traditionally implemented by Prolog, like expert systems, it provides a wide variety of tools and facilities available in Ada. For applications traditionally implemented by Ada, like tools in an advanced environment, the system provides the declarative power of Prolog. In addition to combining these two programming styles, the Ada-Prolog system provides facilities of Ada like concurrency and software system building techniques to Prolog programs. The system also provides a facility to partition the Prolog database which allows easy communication among component systems and allows scoping to be introduced in a database.

The Ada-Prolog system is at the heart of a specification analysis system and has been used for rapid prototyping and other applications. The system has been ported to a Sun-3, a Sequent and an IBM-AT computer. The system brings a host of software engineering techniques to expert systems like formal specifications and clean interfaces.

Further information about the Ada package which implements the Ada-Prolog system can be obtained from the author.

Acknowledgements

The author would like to thank Professor David Luckham at Stanford for his guidance and his support for writing this paper.

This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

References

- [1] *Reference Manual for the ADA Programming Language*. United States Department of Defense. ANSI/MIL-STD-1815A-1983.
- [2] Berkeley/CMU Advanced Environments Workshop, Berkeley, CA. June 1988. Unpublished proceedings.
- [3] Daniel G. Bobrow. If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms. *IEEE Transactions on Software Engineering*, SE-11(11):1401-1408, November 1985.

- [4] W. Chen. *A Theory of Modules for Prolog*. Technical Report, Department of Computer Science, SUNY at Stony Brook, NY, 1986.
- [5] J. Chomicki and N. H. Minsky. Towards a Programming Environment for Large Prolog Programs. In *IEEE Symposium on Logic Programming*, pages 230–241, 1985.
- [6] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM TOPLAS*, 8(1):1–49, January 1986.
- [7] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.
- [8] A. Feuer. Building Libraries in Prolog. In *Proceedings 8th IJCAI, West Germany*, pages 550–552, 1983.
- [9] J. A. Goguen and J. Meseguer. *Equality, Types, Modules and Generics for Logic Programming*. Technical Report, Center for the Study of Language and Information, Stanford University, 1984.
- [10] R. A. Komorowski. QLOG - The Programming Environment for PROLOG in LISP. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*, Academic Press, 1982.
- [11] R. A. Kowalski. Logic as a Database Language. In *Proceedings 3rd British National Conference on Databases*, 1984.
- [12] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *Anna—A Language for Annotating Ada Programs*. Springer-Verlag—Lecture Notes in Computer Science No. 260, July 1987. (Also Stanford University Computer Systems Laboratory Technical Report No. 84-261).
- [13] Walter R. Mann. Implementation of a Specification Analyzer. Technical Report in Preparation.
- [14] K. Morris, J. D. Ullman, and A. VanGelder. NAIL! System Design Overview. In *Proceedings of the 3rd International Conference on Logic Programming, London*, 1986.
- [15] Carlo Muller. Modula Prolog: A Software Development Tool. *IEEE Software*, 39–45, November 1986.
- [16] R. A. O'Keefe. *Prolog and Mixed Language Programming*. Technical Report, Department of Artificial Intelligence, University of Edinburgh, 1984.
- [17] C. Rich and R. C. Waters. *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann Publishers, 1986.
- [18] J. A. Robinson and E. E. Sibert. LOGLISP: Motivation, Design and Implementation. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*, Academic Press, 1982.
- [19] M. Sato and T. Sakurai. *QUTE: A Prolog/Lisp Language for Logic Programming*. Technical Report, Department of Information Science, University of Tokyo, 1983.

- [20] E. Y. Shapiro. Systems Programming in Concurrent Prolog. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah*, January 1984.
- [21] Adrian Walker, Michael McCord, John F. Sowa, and Walter G. Wilson. *Knowledge Systems and Prolog*. Addison-Wesley, 1987.
- [22] R. S. Wallace. *PiL (Prolog in Lisp) Introduction, Implementation, Documentation*. Technical Report, University of Maryland, Computer Science Center, 1983.
- [23] D. H. D. Warren. *Implementing Prolog—Compiling Predicate Logic Programs*. Technical Report 39, D. A. I., Edinburgh, May 1977.